Goals:
- Light both the onboard LEDs with the onboard button.
- Connect two other buttons to Port 1 of the micro.
- Use a polling scheme and subroutines to light the LEDs: one external button should light the Red LED, the other external button should light the Green LED. The onboard button should light both LEDs as before.

Bonus:
Write a C version of your code. Additionally, the LEDs should only stay lit momentarily and then turn off after bit of time. +10

Pre Lab Questions:
- What kind of resistor configuration (Pull-up or Pull-down) is used to connect the onboard button; the external buttons?
- Why do you need a resistor in series with an LED?
- How many subroutines did you use?
In this lab, we will be working with the **Port 1 Registers**. The MSP430 has 2 Ports, but we will only be working with **Port 1** for now. **Port Registers** are 8 bits in size and have pin names **Px.0** through **Px.7**. Each pin corresponds to **Bits 0** through **7**. Below is a brief explanation of the functions of each **Port Register**: 

**Port Registers**

- **P1DIR** — In/Output Pin Direction Selection; **0** = Input, **1** = Output
- **P1IN** — Read Pin State; **0** = Logical Low, **1** = Logical High
  - Register is read-only
- **P1OUT** — Set Pin State; **0** = Logical Low, **1** = Logical High
- **P1REN** — Enable Internal Pin Resistor; **0** = Disconnect, **1** = Connect
  - When the resistor is connected, **P1OUT** determines the Resistor Configuration: **0** = Pulldown, **1** = Pullup
- **P1SEL** — Pin Peripheral Selection 1
- **P1SEL2** — Pin Peripheral Selection 2

**P1DIR** selects the direction of the pin; if the pin is designated as an **input pin**, you can **read** the state of the pin using **P1IN**; conversely, if the pin is designated as an **output pin**, you can **set** the state of the pin using **P1OUT**.

**P1REN** **dis/connects** the internal resistor to a pin according to the pin’s corresponding bit. To set the Resistor Configuration, **Pulldown** or **Pullup**, modify the according bit in **P1OUT**.

**P1SEL** and **P1SEL2** are used to determine which peripheral is connected to a pin. We will not worry about those until we start using peripherals in later labs.

**Port 2** has similar register names, except the names start with **P2** instead of **P1**.
Resistor Configurations

There are two types of Resistor Configurations: **Pulldown** and **Pullup**. The schematics below illustrate the connections as well as the logic flow:
Reading and Writing to Port Registers

Reading and writing to can be done with a few choice Assembly operations. Below is a table summarizing the most useful operations for writing to a Port Register; for your convenience, the equivalent operations in C are also listed:

<table>
<thead>
<tr>
<th>ASSEMBLY</th>
<th>C</th>
<th>EXAMPLE</th>
<th>NOTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIS</td>
<td></td>
<td>=</td>
<td>`bis.b #00001111, &amp;P1OUT P1OUT</td>
</tr>
<tr>
<td>BIC</td>
<td>&amp;</td>
<td>= (~…)</td>
<td><code>bic.b #11110000, &amp;P1OUT P1OUT &amp;= ~(0b11110000);</code></td>
</tr>
<tr>
<td>MOV</td>
<td>=</td>
<td><code>mov.b #01000001, &amp;P1OUT P1OUT &amp;= ~(0b01000001);</code></td>
<td>Overwrite P1OUT.</td>
</tr>
<tr>
<td>XOR</td>
<td>^=</td>
<td><code>xor.b #01000001, &amp;P1OUT P1OUT ^= 0b01000001;</code></td>
<td>Toggle LED bits in P1OUT.</td>
</tr>
<tr>
<td>CMP</td>
<td>–</td>
<td><code>cmp.b #00010000, &amp;P1IN if( !(P1IN – BIT3) ){ ... }</code></td>
<td>Check if P1IN is equal to BIT3.</td>
</tr>
<tr>
<td>BIT</td>
<td>&amp;</td>
<td><code>bit.b #00001000, &amp;P1IN if( (P1IN &amp; BIT3) == BIT3){ ... }</code></td>
<td>Check if BIT3 is set in P1IN.</td>
</tr>
</tbody>
</table>

Note that both CMP and BIT do not store the result in any register, they merely set the Z or C Status Bits, respectively.

Reading a Port Register can be done with the same operations as above, only with the Port Register as a source register instead of a destination register, or equivalently having the Port Register on the right-hand side of the ‘=’ sign in C. See below for some examples:

```assembly
mov.b &P1IN, R4 ; Store state of input pins into R4.
InputPinSTATE = P1IN; // Store state of input pins into a meaningful variable.

bis.b &P1IN, &P1OUT ; Set bits in P1OUT according to high pins of P1IN.
P1OUT |= P1IN; // Set bits in P1OUT according to high pins of P1IN.
```
In practice, we would read bits in a **Port Register** and use the data in conditional statements. As an example, consider a button connected to P1.5. Assuming we set up the **Port Registers** correctly, we can test the state of the button as such:

```assembly
bit.b  #00100000, &P1IN;  // Test the state of P1IN; if result is greater than 1, …
jc     LEDON            // ... the C Status Bit will be set and we jump to label.
;...
LEDON
bis.b  #01000000, &P1OUT; // Turn on the Green LED.
;...

if( (P1IN & BIT5) == BIT5) // Test the state of P1IN; if BIT5 is set...
{
    P1OUT |= BIT6;        // ... turn on the Green LED.
}
```
Subroutines work in a similar fashion. In Assembly, we use the **CALL** and **RET** operations. See below for an example:

```assembly
MAIN
    ;…
    call    #SUBROUTINE ; Call a Subroutine.
    ;…
    jmp     MAIN

SUBROUTINE:
    ;…
    ret    ; Return to instruction after Subroutine call.
```

```c
int main(void)
{
    //…
    while(1)
    {
        //…
        Subroutine(); // Call a Subroutine. //…
    }
}

void Subroutine(void)
{
    //…
    return; // Return to instruction after Subroutine call.
}
```

**Notes:**

- The micro considers any voltage below 0.8V as a Logical 0, while any voltage above 2.4V as a Logical 1. Voltages in **between** 0.8V and 2.4V will produce an **unpredictable result**.
- Become comfortable with both Resistor Configurations, different sensors and devices will require one or the other. **Note that the state of the button/switch of each configuration produces the opposite logical value.**
- The internal resistor is useful if you need to physically ground or keep a pin powered. Remember that the pins on the micro are very small and close together; as such it is possible for one pin to induce a current or voltage on a near-by pin, causing and undesired logic level on the near-by pin. Grounding or powering the pin using the internal resistor helps eliminate this problem.
- Note the ‘#’ sign in front of the label when using **CALL** ins Assembly.
The Pullup Resistor Configuration is best for the onboard button while the Pulldown Resistor Configuration is probably best for the external buttons just from a logical point of view, although either configuration will work.

It is possible to complete this lab without the use of any subroutines, however, it would be good practice to employ them as they will be essential later. Take a look at the flow chart below if you are having trouble writing your program:
**Port Register Demo:**

User the interactive demo below to familiarize yourself with the basic Operations and Port Registers.
Lab Notebook Questions:

-Briefly explain the four Port 1 Registers that we will be using in this lab.
-Explain how the following commands work: bit.b, bic.b and bis.b.
-Briefly explain the purpose of branching and give examples of how they are implemented in assembly.
-What is the difference between using jump operations with labels and CALL and RET?
**Port I/O**

**Frequently Asked Questions**

*What size resistors should I use with my LEDs?*

It depends on the LED. Most LEDs will come with a datasheet that will tell you the correct size or resistor to use, but a good place to start is with any resistor between 500Ω and 1kΩ.

*Can I modify P1IN so simulate a button being pushed?*

No. **P1IN** is read-only. Attempting to write to the register will have no effect. You will have to physically change the state of the pins. Alternatively, you can temporally change the condition of the desired code (whatever code you are expecting the button/pin to affect) to always be true or false in the meantime.

*How do I connect my switch!?*

Checkout the diagram below if you are having trouble connecting your switch: